Softwareentwicklung und Betriebssysteme

Vorbereitung für die Summer School Dritteinsteiger 2019

Der Teil Softwareentwicklung und Betriebssysteme der Summer School Dritteinsteiger 2019 beginnt am 8.7.2019 um 16:00.

In dieser Lehrveranstaltung (LV) werden die wichtigsten Inhalte aus Softwareentwicklung, Algorithmen und Datenstrukturen und Betriebssysteme aus dem 1. und 2. Semester Informationstechnik und Systemmanagement wiederholt. Die Summer School dient nur der Wiederholung und Auffrischung und kann das *eigene Vorwissen* sowie *selbstständiges Lernen und Üben* nicht ersetzten.

Lehrveranstaltungsübersicht

Die LV besteht aus 15 Einheiten (je 45 min) und 2 Einheiten Klausur (insgesamt 90 min). Die Gesamtnote setzt sich aus selbstständig, zu Hause vorbereiteten Übungsaufgaben, die in der LV präsentiert werden (30%) und einer abschließenden schriftlichen Prüfung (70%) in den letzten beiden Einheiten zusammen, wobei beide Teile positiv sein müssen.

Alle Unterlagen zur LV finden Sie online unter `https://www.en-trust.at/knirsch/teaching/`:

- Die Folien zur LV finden Sie im Dokument *Folien gesamt*.
- Die klausurrelevanten Inhalte der LV finden Sie im Dokument *Klausurvorbereitung*.
- Freiwillige Übungsaufgaben zur Klausurvorbereitung finden Sie im Dokument *Übungsaufgaben*.

Bereiten Sie sich mit diesen Unterlagen vor, um in der LV gezielt Fragen zu stellen bzw. Schwerpunkte zur Wiederholung zu definieren.

Literatur: Wolf, J.: *C von A bis Z – Das umfassende Handbuch*, Galileo Computing, 2005. Als Open Book frei zugänglich `http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/`

Vorbereitungsaufgaben

Bitte bereiten Sie bis zum 16.07.2019 *mindestens zwei* Aufgaben Ihrer Wahl aus den folgenden Übungsaufgaben in C vor.

Die Aufgaben unterscheiden sich in Schwierigkeit und Fokus. Wählen Sie jene Aufgaben, die Aspekte behandeln bei denen Sie bei sich selbst den größten Übungsbedarf sehen. Es steht

Ihnen frei, über die geforderte Aufgabenstellung hinaus Optimierungen und Erweiterungen umzusetzen.

Zur Umsetzung Ihrer Aufgaben können Sie die C-Standardbibliothek verwenden. Greifen Sie nicht auf externe Bibliotheken oder Code-Ausschnitte zurück, sondern versuchen Sie zu Übungszwecken das Programme vollständig selbst zu implementieren. Kennzeichnen Sie etwaige externe Quellen, übernommene Code-Ausschnitte (auch von KollegInnen) oder Hilfestellungen mit Kommentaren im Quellcode.

Die Programme sollen fehlerfrei kompilieren, die geforderte Aufgabenstellung lösen und Sie sollen in der Lage sein, Ihre Implementierung sowie den gewählten Lösungsweg vor Ihren Kollegen in der Lehrveranstaltung zu präsentieren. Die Präsentation und die Qualität der Programme fließen zu 30% in die Gesamtnote ein.

Übungsaufgaben

Übungsaufgabe 1

Schreiben Sie ein C-Programm *sortbookdata* zum Einlesen, Sortieren und Speichern der Datei *bookdata.xml*. Die Daten sollen in einer geeigneten Datenstruktur zwischengespeichert werden. Das Programm nimmt drei Argumente entgegen: Quelldatei, Zieldatei, Sortierung. Sortierung kann *Titel*, oder *Autor* sein. Die Zieldatei soll nach dem im Feld Sortierung angegebenen Attribut sortiert sein. Schätzen Sie die Komplexität des von Ihnen implementierten Sortieralgorithmus in Groß-O-Notation ab.

Beispielaufruf:

```
> sortbookdata bookdata.xml out.xml Autor
```

Dieser Aufruf liest die Datei *bookdata.xml* ein, sortiert die Daten nach dem Autor und speichert das Ergebnis als *out.xml*.

Hinweise:

- Unter `http://www.users.fh-salzburg.ac.at/~fhs31108/teaching/bookdata.xml` finden Sie eine Datei mit Buch-Daten.
- Definieren Sie eine geeignete `structure`, die alle Informationen zu einem Buch aufnimmt und verwenden Sie z.B. eine doppelt verkettete Liste zum Speichern der Daten.
- Die Sortierung kann mit dem Quicksort-Algorithmus aus der C-Standardbibliothek oder mit einer eigenen Implementierung erfolgen.

Übungsaufgabe 2

Schreiben Sie ein C-Programm *feistel* zur Ver- und Entschlüsselung einer Buchstabenfolge mit der Feistel-Chiffre. Diese Chiffre ist ein einfaches symmetrisches Verfahren. Der Klartext M wird in zwei gleich große Blöcke $M = (L_1, R_1)$ aufgeteilt, jede Runde hat einen Rundenschlüssel K_i. Die $i = 1 \dots n$ Runden für die Verschlüsselung sind wie folgt:

$$L_{i+1} = R_i$$

$$R_{i+1} = L_i \oplus F(R_i, K_i)$$

Recherchieren Sie in der Literatur eine geeignete Funktion F sowie den Algorithmus zur Entschlüsselung.

Beispielaufrufe:

```
> feistel -enc KLARTEXT GEHEIMER_SCHLUESSEL
> feistel -dec CHIFFRE GEHEIMER_SCHLUESSEL
```

Übungsaufgabe 3

Schreiben Sie ein C-Programm *calc*, das die folgende über EBNF definierte Grammatik parsen und das Ergebnis eines Textes in dieser Grammatik berechnen kann. Lesen Sie dazu einen Text in dieser Grammatik über eine Datei oder über `stdin` ein; die Ausgabe kann auf `stdout` erfolgen.

Grammatik:

```
DIGIT = "0" | "1" | ... | "9";
SIGN = "+" | "-";
NUM = [SIGN] DIGIT {DIGIT};
REGISTER = "A" | "B" | ... | "Z";

ADD = "ADD " REGISTER " " (NUM | REGISTER) " " (NUM | REGISTER);
SUB = "SUB " REGISTER " " (NUM | REGISTER) " " (NUM | REGISTER);
MUL = "MUL " REGISTER " " (NUM | REGISTER) " " (NUM | REGISTER);
DSP = "DSP " REGISTER;
COMMAND = (ADD | SUB | MUL | DSP) "\n";
PROGRAM = {COMMAND};
```

Beispiel eines gültigen Textes (Kommentare nur zur Erklärung):

```
ADD A -10 5 // Register A hat hat nach Ausführung den Wert -5
```

```
MUL B A 3   // Register B hat nach Ausführung den Wert -15
DSP B       // Ausgabe: -15
```

Hinweise:

- Kommandos müssen exakt der oben angegebenen Grammatik folgen, d.h. auch die Anzahl und Position von Leerzeichen. Geben Sie bei ungültigen Kommandos eine Fehlermeldung aus oder brechen Sie die Ausführung ab.
- Initialisieren Sie alle Register A-Z bei Programmstart mit 0.
- Um ein Kommando zu erkennen, zerlegen Sie es entsprechend der Grammatik: 3 Zeichen Kommando (ADD, SUB, etc.), 1 Leerzeichen, Symbol "A"-"Z" wenn Register, sonst erkennen der Zahl, usw.

Übungsaufgabe 4

Schreiben Sie ein C-Programm *exprand*, das N exponentialverteilte Pseudozufallszahlen für ein gegebenes λ erzeugt und diese zeilenweise in eine Datei schreibt. Verwenden Sie dazu die Inversionsmethode und lesen sie N und λ als Kommandozeilenargumente ein.

Die Exponentialverteilung hat folgende Verteilungsfunktion:

$$F(x) = 1 - \exp(\lambda x)$$

Überprüfen Sie, ob die Pseudozufallszahlen der gewünschten Verteilung folgen, indem Sie die erzeugte Datei in (z.B.) Matlab einlesen und die Verteilung visualisieren (ohne statistischen Test).

Hinweise:

- Die Inversionsmethode erlaubt es bestimmte Wahrscheinlichkeitsverteilungen aus gleichverteilten (Pseudo-)Zufallszahlen zu erzeugen. Verwenden Sie den Pseudozufallszahlengenerator `rand()`, um Pseudozufallszahlen zwischen 0 und 1, sowie die inverse Verteilungsfunktion $F^{-1}(x)$, um daraus exponentialverteilte Pseudozufallszahlen zu erzeugen.
- Geben Sie die erzeugten N Pseudozufallszahlen auf `stdout` aus, um das Ergebnis mit einem anderen Programm zu visualisieren.

Übungsaufgabe 5

Schreiben Sie ein C-Programm *hashtest*, das eine Hash-Tabelle mit N einträgen für Tupel aus (österreichischer) Postleitzahl und Ortsname implementiert. Die Postleitzahl wird dabei als Schlüssel verwendet. Recherchieren Sie in der Literatur und entwerfen Sie eine geeignete Hash-Funktion $H(\cdot)$, um eine Postleitzahl auf einen Index $1, \ldots, N$ abzubilden. Lesen Sie eine CSV-Datei mit Postleitzahlen ein und bauen Sie eine Hash-Tabelle mit $N = 1000$ Einträgen auf. Probieren Sie verschiedene Varianten der Funktion H und bewerten Sie deren Qualität anhand der Verteilung der Einträge und der Anzahl der auftretenden Kollisionen.

Beispiel der CSV-Datei:

```
5020;Salzburg
1010;Wien
...
```

Beispiel einer Hash-Funktion:

$$H(m) = m \mod 997$$

Zuordnung bei Verwendung dieser Hash-Funktion:

```
1:    -
2:    -
...
13:   1010 -> Wien
...
35:   5020 -> Salzburg
...
1000: -
```

Zählen Sie bei Kollisionen nur deren Auftreten, um die Qualität der Hash-Funktion zu bewerten. Implementieren Sie als optionale Erweiterung eine vollständige Behandlung von Kollisionen (z.B. mittles Erstellung von Buckets). Ihr Programm kann beispielsweise die folgenden Funktionen abbilden:

- `void init(const int N)` – initialisieren einer Hash-Tabelle für N Elemente
- `void put(const int key, const void* value)` – Ablegen eines Wertes zu einem Schlüssel in der Hash-Tabelle
- `void* get(const int key)` – Lesen eines Wertes aus der Hash-Tabelle für einen gegebenen Schlüssel

Hinweise:

- Eine Liste der österreichischen Postleitzahlen finden Sie unter `https://www.post.at/geschaeftlich_werben_produkte_und_services_adressen_postlexikon.php`. Laden Sie das PLZ-Verzeichnis als Excel-Datei herunter und speichern Sie die ersten beiden Spalten als CSV-Datei.
- Eine zusätzliche Erweiterung ist eine komplett generische Variante der Hash-Tabelle. Schlüssel sind dann nicht nur vom Typ `int`, sondern vom Typ `void*`, d.h. können beliebigen Typs sein. Die Funktion $H(\cdot)$ muss entsprechend angepasst werden.

Übungsaufgabe 6

Die Vernam-Chiffre ist ein einfaches und informationstheoretisch sicheres symmetrisches Verschlüsselungsverfahren. Zur Verschlüsselung wird ein Klartext $m \in \mathbb{Z}_n$ zu einer gleich langen, nur einmalig verwendeten und zufälligen Folge $r \in \mathbb{Z}_n$ addiert:

$$c = m + r \mod n.$$

Der verschlüsselte Text hat exakt die gleiche Länge wie der Klartext. Für die Entschlüsselung der Nachricht muss der Empfänger über die selbe zufällige Folge verfügen und berechnet:

$$m = c - r \mod n.$$

Wichtig: Die im folgenden umzusetzende Implementierung wird die informationstheoretische Sicherheit nicht gewährleisten. Recherchieren Sie zunächst in der Literatur welche Anforderungen für eine solche Sicherheitsgarantie erforderlich sind und inwiefern ihr Programm davon abweichen wird.

Schreiben Sie ein C-Programm *vernam*, das eine praktische Variante der Vernam-Chiffre umsetzt. Implementieren Sie die folgenden Eigenschaften:

1. Über drei Kommandozeilenparameter kann dem Programm eine zu verschlüsselnde bzw. zu entschlüsselnde Textdatei, eine Textdatei die den Schlüssel enthält, sowie ein Parameter für den Modus (`enc` für Verschlüsselung, `dec` für Entschlüsselung), mitgegeben werden. Beispiel für den Aufruf: `vernam eingabe.txt schluessel.txt enc`
2. Für eine einfachere Implementierung werden nur Texte mit den Zeichen A-Z unterstützt, d.h. $m, r \in \{A, \ldots, Z\}^*$ (ASCII 0x41 bis 0x5A). Ersetzen Sie Leerzeichen im Klartext durch das Zeichen *X*.
3. Berechnen Sie die Ver- bzw. Entschlüsselung wie oben angegeben und schreiben Sie das Ergebnis in eine Ausgabedatei.

4. Freiwillige Zusatzaufgabe: Zur besseren Lesbarkeit sind die Daten der Ausgabedatei und des Schlüssels in 5 Zeichen lange Blöcke mit Leerzeichen getrennt gruppiert. Diese Leerzeichen sind nicht Teil des Klartextes bzw. des Schlüssels und dienen nur der visuellen Gruppierung. Beachten Sie jedoch, dass der Klartext dazu eine Länge haben muss, die ganzzahlig durch 5 teilbar ist, d.h. $|m| \mod 5 = 0$. Ergänzen Sie ggf. den Klartext am Ende mit dem Zeichen X, um eine Länge zu erreichen die ein ganzzahliges Vielfaches von 5 darstellt.

Beispiel:

1. Eingabedatei im Originaltext `DAS IST EIN GEHEIMER KLARTEXT` → Aufbereitet für Verschlüsselung `DASXISTXEINXGEHEIMERXKLARTEXTX` (Leerzeichen durch X ersetzt, ein X am Ende für die Teilbarkeit durch 5)
2. Schlüsseldatei: `HFSFD SJFLK SDFJO QWENQ QEEQU USDGG`
3. Ausgabedatei: `KFKCL KCCPS FALNV UEQRH NOPQL NWAZD`

Hinweise:

- Implementieren Sie eine entsprechende Fehlerbehandlung, etwa wenn Klartext und Schlüssel bzw. verschlüsselter Text und Schlüssel nicht dieselbe Länge haben.
- Sehen Sie auch die Möglichkeit vor, mit Ihrem Programm bei Bedarf einen ausreichend langen, pseudozufälligen Schlüssel zu erzeugen.
- Unter `http://www.users.fh-salzburg.ac.at/~fhs31108/teaching/` finden Sie die Dateien `ausgabe.txt` und `schluessel.txt`, die einen verschlüsselten Beispieltext bzw. Schlüssel beinhalten.
- Das Verschlüsselungsverfahren ist auch als *One-Time-Pad* bekannt und wird ursprünglich zur Anwendung auf eine Bitfolge beschrieben. Dabei kann die Operation *XOR* zur Verknüpfung von Klartext/Schlüsseltext und Schlüssel verwendet werden. Dies ist ein Spezialfall der oben beschriebenen Modulo-n-Operation, wobei sich n aus den Elementen im Klartextalphabet ergibt (hier 26).